Swift (*pol. jerzyk*)

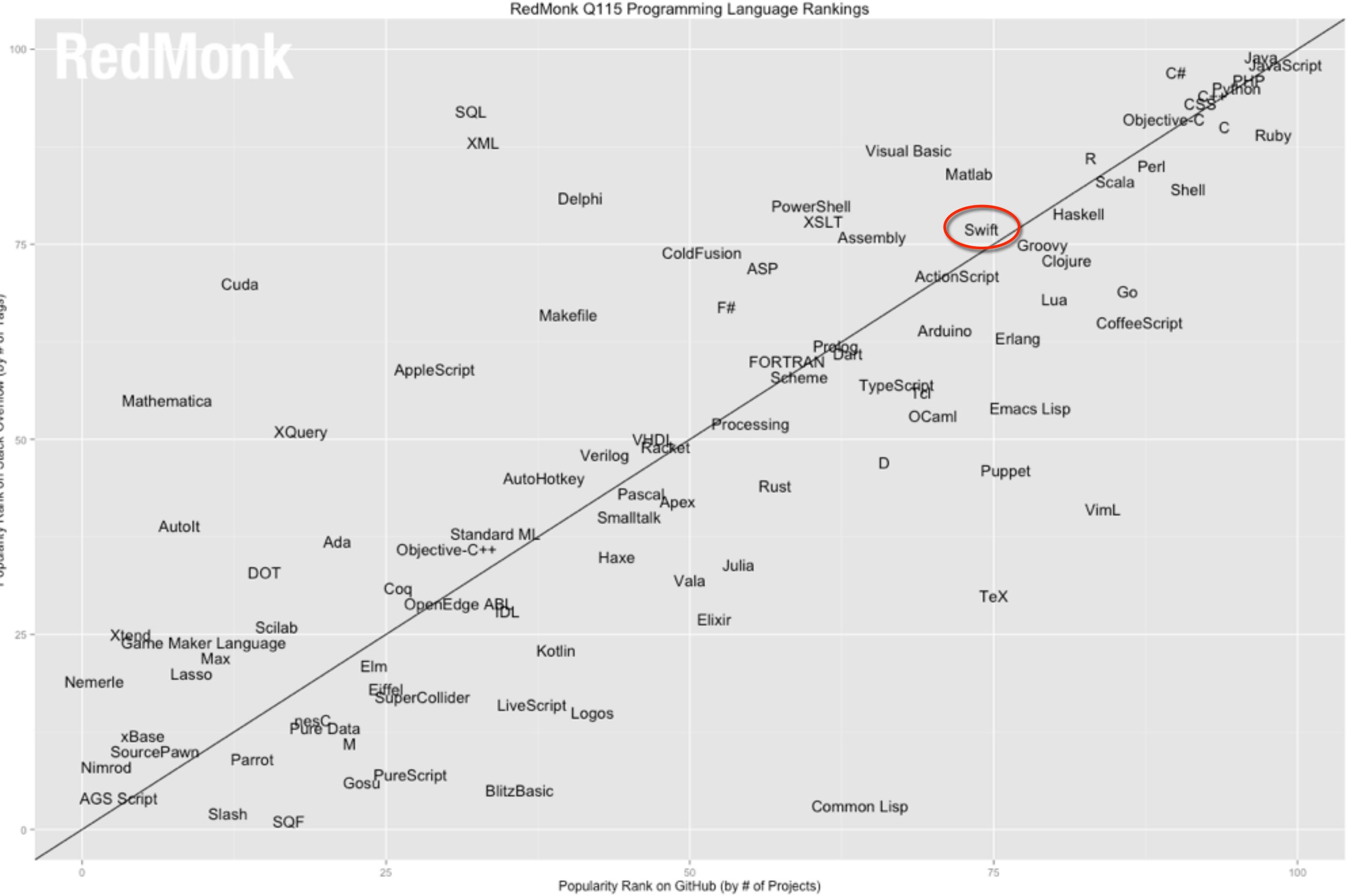
- nowy język programowania zaprezentowany latem 2014 r. (prace od 2010 r.)
- przeznaczony do programowania zarówno pod iOS jak i Mac OS X
- bazuje na logice Objective-C bez kompatybilności z C

RedMonk Q115 Programming Language Rankings

RedMonk

Popularity Rank on Stack Overflow (by # of Tags)

Popularity Rank on GitHub (by # of Projects)



Podobieństwa do Objective-C

- typy podstawowe
- operatory
- użycie nawiasów klamrowych `{ }`
- użycie nawiasów kwadratowych `[ ]` (tablice)
- operator przypisania =, porównania ==
- podobne instrukcje (pętle, warunkowe)
- dziedziczenie metod klas i instancji
- słowo kluczowe `self`

Różnice vs Objective-C

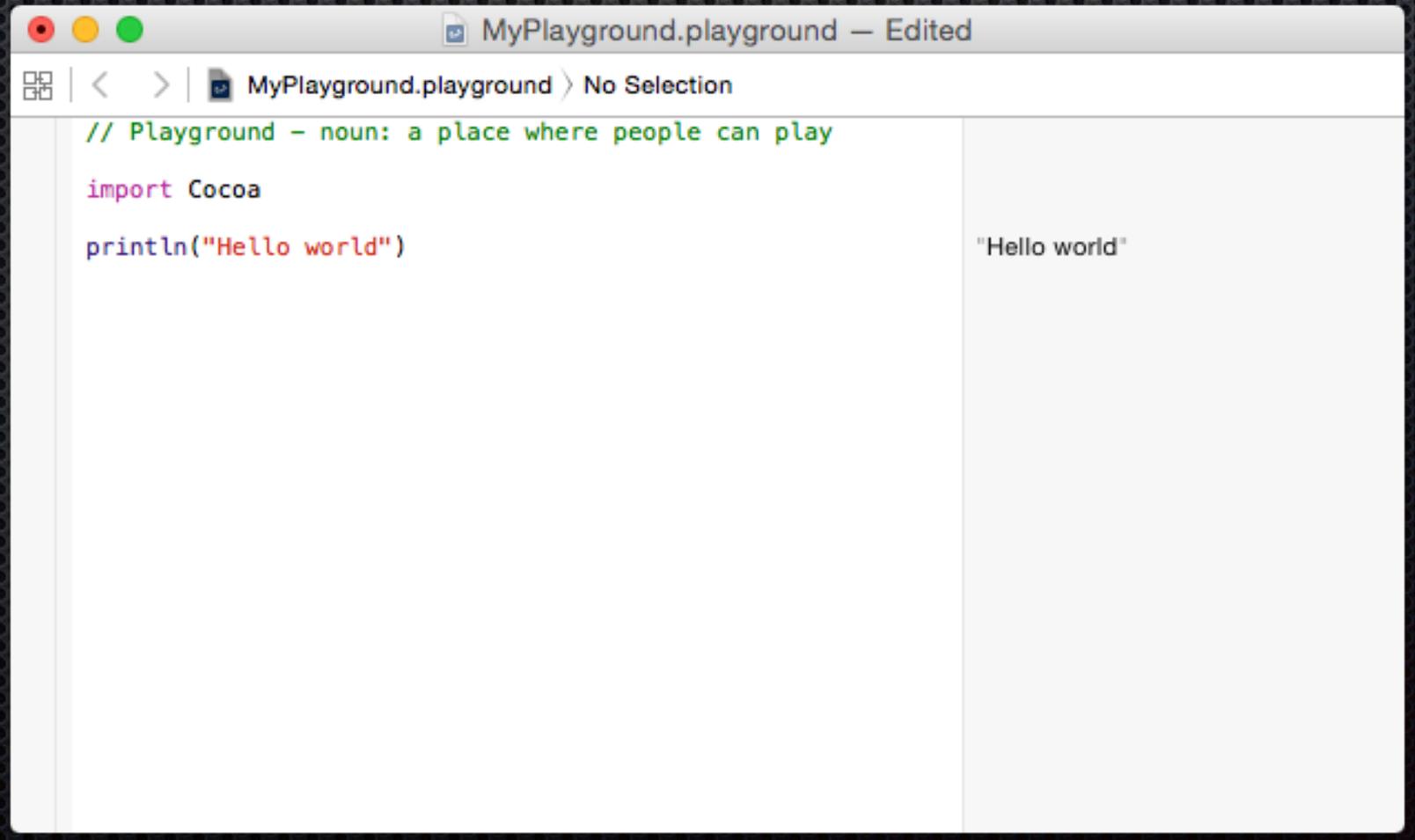
- nie trzeba używać ;
- brak plików nagłówkowych
- silne typowanie
- przeciążanie operatorów
- nowy operator === porównujący obiekty
- pełne wsparcie Unicode w string
- brak obsługi wyjątków
- inferencja typów
- programowanie generyczne
- typy funkcyjne

Różnice vs Objective-C

- domyślnie nie ma wskaźników
- przypisanie nie zwraca wartości
- brak konieczności używania `break` w instrukcji `switch` (instrukcja `fallthrough`)
- zmienne i stałe są zawsze inicjalizowane
- przepełnienie zakresu jest wykrywane, chyba, że użyjemy specjalnych operatorów, np. `&+`, `&-`

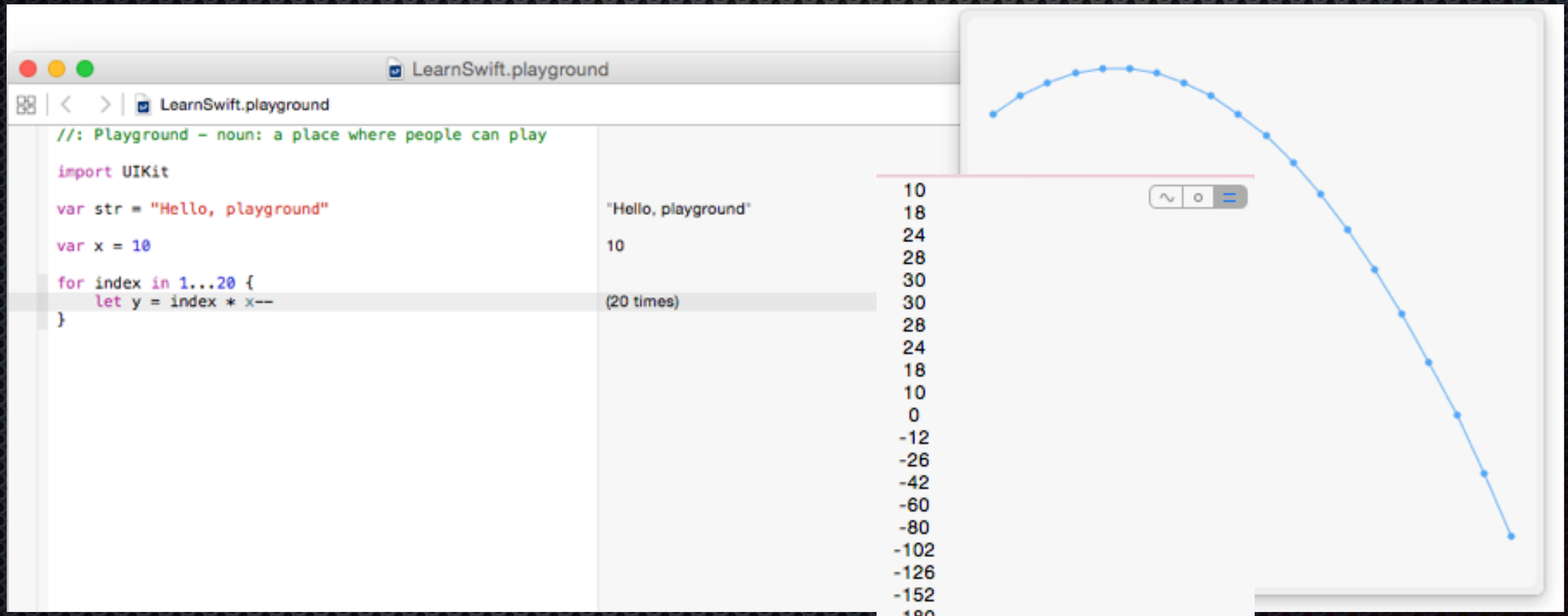
Playground

- xCode 6 zawiera moduł Playground
- służy on do sprawdzania na żywo jak działa kod Swift



Playground

- Można na różne sposoby wizualizować działanie kodu

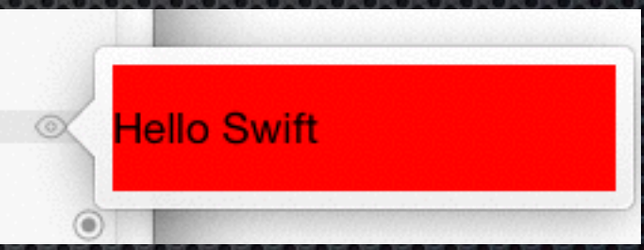


Playground

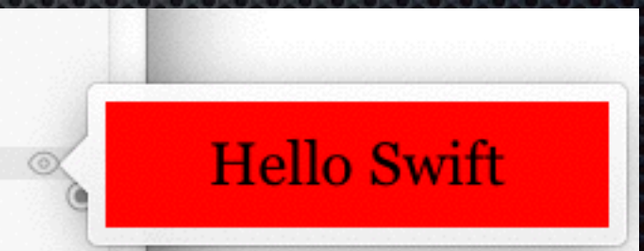
- Można korzystać z większości API, np. UIKit

```
let myLabel = UILabel(frame: CGRectMake(0, 0, 200, 50))
myLabel.backgroundColor = UIColor.redColor()
myLabel.text = "Hello Swift"
myLabel.textAlignment = .Center
myLabel.font = UIFont(name: "Georgia", size: 24)
myLabel
```

```
let myLabel = UILabel(frame: CGRectMake(0, 0, 200, 50))
myLabel.backgroundColor = UIColor.redColor()
myLabel.text = "Hello Swift"
myLabel.textAlignment = .Center
myLabel.font = UIFont(name: "Georgia", size: 24)
myLabel
```




```
let myLabel = UILabel(frame: CGRectMake(0, 0, 200, 50))
myLabel.backgroundColor = UIColor.redColor()
myLabel.text = "Hello Swift"
myLabel.textAlignment = .Center
myLabel.font = UIFont(name: "Georgia", size: 24)
myLabel
```



Zmienne i stałe

- Słowa kluczowe let i var
  - let definiuje stałą
  - var definiuje zmienną
- podstawienie wartości określa typ zmiennej/stałej dzięki inferencji typów

```
var myVariable = 42                    42
myVariable = 50                        50
let myConstant = 42                    42
```

```
let implicitInteger = 70               70
let implicitDouble = 70.0              70.0
let explicitDouble: Double = 70        70.0
```

Zmienne i stałe

- Nazwy zmiennych mogą zawierać dowolne znaki Unicode oprócz:
  - spacji
  - symboli matematycznych
  - linii i prostokątów
- Nie mogą się zaczynać od cyfry

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Typy podstawowe

- `Int8`
- `UInt8`
- `Int` (Int32, Int64)
- `UInt` (UInt32, UInt64)
- `Double` (64 bity - 15 cyfr)
- `Float` (32 bity - 6 cyfr)
- `String`
- `Bool` (true lub false)

Typy podstawowe

- Dla których ma to sens mają własności max i min

```
let tooBig: Int8 = Int8.max
```

- Można stosować aliasy nazw typów

```
typealias AudioSample = UInt16
```

Operatory

- Podstawowe operatory: +,-,*,/,% (modulo)
- Operatory złożone: +=, -=, *=, /=, %=, &&, ||
- Operatory inkrementacyjne: ++, —
- Operatory porównania: ==, >, <, >=, <=, !=
- Operatory zakresu: x..y, x..<y
- Operatory bitowe: &, |, ^, <<, >>, &=, |=, ^=, <<=, >>=

Operatory typów

- Operator `is` sprawdza, czy obiekt jest danego typu i ma wartość boolowską
- Operator `as?` rzutuje obiekt na typ opcjonalny, a gdy jest to niemożliwe podstawia `nil`
- Operator `as!` wymusza rzutowanie na typ, a jeżeli jest niemożliwe to występuje błąd wykonania
- Operator `as` można stosować przy rzutowaniu w górę

Zmienne i stałe

- wartości nigdy nie są domyślnie konwertowane do innych typów - trzeba stworzyć nową wartość

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

```
"The width is "
94
"The width is 94"
```

- wartości w stringach można osadzać łatwiej

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces
    of fruit."
```

```
3
5
"I have 3 apples."
"I have 8 pieces of fruit."
```

- typy opcjonalne (nullowe) - dodajemy `?` wtedy zmienna może mieć wartość `nil`

```
var optionalString: String? = "Hello"

optionalString == nil
```

Tablice i słowniki

- Deklaruje się je za pomocą nawiasów kwadratowych
- Dostęp do elementów też przez nawiasy

```
var shoppingList = ["catfish", "water", "tulips",
    "blue paint"]
shoppingList[1] = "bottle of water"
```

```
["catfish", "water", "tulips", "blue...
"bottle of water"
```

- Słowniki klucz:wartość

```
var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

```
["Kaylee": "Mechanic", "Malcolm"...
{Some "Public Relations"}
```

- Aby utworzyć puste korzystamy ze składni konstruktorów

```
let emptyArray = [String]()
let emptyDictionary = [String: Float]()
```

```
0 elements
0 key/value pairs
```

```
shoppingList = []
occupations = [:]
```

```
0 elements
0 key/value pairs
```

Krotki

- Wartości można grupować w krotki (tuples)

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and
        equals (404, "Not Found")
```

- Krotka może zawierać dowolną kombinację typów

- Krotkę można zdekomponować na składowe

```
let (statusCode, statusMessage) = http404Error
println("The status code is \(statusCode)")
// prints "The status code is 404"
println("The status message is \(statusMessage)")
// prints "The status message is Not Found"
```

Krotki

- niepotrzebne wartości można zastąpić podkreśleniem

```
let (justTheStatusCode, _) = http404Error
println("The status code is \
        (justTheStatusCode)")
// prints "The status code is 404"
```

- do składowych można się odwołać przez indeks

```
println("The status code is \(http404Error.0)")
// prints "The status code is 404"
println("The status message is \
        (http404Error.1)")
// prints "The status message is Not Found"
```

Krotki

- składowe mogą mieć swoje etykiety

```
let http200Status = (statusCode: 200,
        description: "OK")
```

```
println("The status code is \
        (http200Status.statusCode)")
// prints "The status code is 200"
println("The status message is \
        (http200Status.description)")
// prints "The status message is OK"
```

Instrukcje

- instrukcje warunkowe `if` i `switch`
  - warunek musi być zawsze boolowski
- pętle `for-in`, `for`, `while`, `do-while`

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
println(teamScore)
```

```
[75, 43, 103, 87, 12]
0


(3 times)

(2 times)


"11"
```

Instrukcja warunkowa

- Jeżeli chcemy sprawdzić, czy zmienna ma wartość w instrukcji warunkowej można wykorzystać operator let

```
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

```
{Some "John Appleseed"}
"Hello!"

"Hello, John Appleseed"
```

```
var optionalName: String? = nil
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

```
nil
"Hello!"
```

Instrukcja `switch`

- Nie jest ograniczona do typów prostych
- Nie jest ograniczona do równości
- Musi być wyczerpująca - klauzula `default`

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(x)?"
default:
    let vegetableComment = "Everything tastes good in soup."
}
```

"red pepper"

"Is it a spicy red pepper?"

```
var temperature = 83

switch (temperature)
{
case 0...49:
    println("Cold")
case 50...79:
    println("Warm")
case 80...110:
    println("Hot")
default:
    println("Temperature out of range")
}
```

83

"Hot"

Instrukcja `switch`

- W przeciwieństwie do innych języków Swift automatycznie kończy **`switch`** po znalezieniu pasującego przypadku
- Można użyć **`break`**, gdy tego potrzeba
- Można to zmienić instrukcją **`fallthrough`**

```
var temperature = 60
switch (temperature)
{
case 0...49 where temperature % 2 == 0:
    println("Cold and even")
    fallthrough
case 50...79 where temperature % 2 == 0:
    println("Warm and even")
    fallthrough
case 80...110 where temperature % 2 == 0:
    println("Hot and even")
    fallthrough
default:
    println("Temperature out of range or odd")
}
```

```
60

"Warm and even"

"Hot and even"

"Temperature out of range or odd"
```

Instrukcja `for-in`

- Umożliwia iterowanie po tablicach i słownikach

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
println(largest)
```

```
["Prime": [2, 3, 5, 7, 11,...
0
(8 times)
"25"
```

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
var k = "None"
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
            k=kind
        }
    }
}
println("\(largest) in \(k)")
```

```
["Prime": [2, 3, 5, 7, 11,...
0
"None"
(8 times)
(8 times)
"25 in Square"
```

Pętle

- `while`, `do-while`

```
var n = 2
while n < 100 {
    n = n * 2
}
println(n)

var m = 2
do {
    m = m * 2
} while m < 100
println(m)
```

```
2

(6 times)

"128"

2

(6 times)

"128"
```

- `for`

```
var firstForLoop = 0
for i in 0..<4 {
    firstForLoop += i
}
println(firstForLoop)

var secondForLoop = 0
for var i = 0; i < 4; ++i {
    secondForLoop += i
}
println(secondForLoop)
```

```
0

(4 times)

"6"

0

(4 times)

"6"
```

`..<`

```
var firstForLoop = 0
for i in 0...4 {
    firstForLoop += i
}
println(firstForLoop)
```

```
0

(5 times)

"10"
```

`...`

Pętle

- przerwanie pętli - break

```
var j = 10

for var i = 0; i < 100; ++i
{
    j += j

    if j > 100 {
        break
    }

    println("j = \(j)")
}
```

- zakończenie bieżącego przebiegu - continue

```
var i = 1

while i < 20
{
    ++i

    if (i % 2) != 0 {
        continue
    }

    println("i = \(i)")
}
```

Funkcje

- Do deklaracji funkcji służy słowo kluczowe func
- Argumenty oddzielone przecinkami, poprzedzone etykietami i dwukropkiem
- Typ funkcji po argumentach oddzielony ->

```
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
greet("Bob", "Tuesday")
```

"Hello Bob, today is Tues...

"Hello Bob, today is Tues...

Funkcje

- Argumenty funkcji mogą mieć etykiety
- Trzeba z nich wtedy korzystać w trakcie wywołania

```
func buildMessage(#name: String, #count: Int) -> String {
    return("\(name), you are customer number \(count)")
}

var message = buildMessage(name: "John", count: 100)
```

```
"John, you are customer number 100"


"John, you are customer number 100"
```

- Argumenty mogą mieć wartości domyślne

```
func buildMessage(count: Int, name: String = "Customer") -> String {
    return ("\(name), you are customer number \(count)")
}

var message = buildMessage(100)
println(message)

message=buildMessage(50, name: "John")
```

```
(2 times)


"Customer, you are customer number 100"
"Customer, you are customer number 100"

"John, you are customer number 50"
```

Funkcje

- Funkcja może mieć wiele wartości, identyfikowanych przez etykiety lub indeks

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}
let statistics = calculateStatistics([5, 3, 100, 3, 9])
println(statistics.sum)
println(statistics.2)
```

```
5
5
0


100

3

(5 times)


(.0 3, .1 100, .2 120)

(.0 3, .1 100, .2 120)
"120"
"120"
```

Funkcje

- Liczba argumentów funkcji może być zmienna pod warunkiem, że są tego samego typu

```
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}
sumOf()
sumOf(42, 597, 12)
sumOf(10,20,30,40,50,60,70,90)
```

(3 times)

(11 times)

(3 times)

0
651
370

Funkcje

- Domyślnie argumenty są przekazywane jako stałe - nie można ich modyfikować
- Można to zmienić deklarując argument jako zmienną
- Nie zmienia to wartości poza funkcją

```
func calcuateArea (var length: Float, var width: Float) -> Float {
    length = length * 2.54
    width = width * 2.54
    return length * width
}

println(calcuateArea(10, 20))
```

```
25.4
50.8
1290.32


"1290.32"
```

Funkcje

- Argumenty wyjściowe można uzyskać przez słowo kluczowe `inout`
- Przekazywany argument musi być wtedy poprzedzony **&**

```
var myValue = 10

func doubleValue (inout value: Int) -> Int {
    value += value
    return(value)
}

println("Before function call myValue = \(myValue)")

println("doubleValue call returns \(doubleValue(&myValue))")

println("After function call myValue = \(myValue)")
```

```
10


20
20


"Before function call myValue = 10"

"doubleValue call returns 20"

"After function call myValue = 20"
```

Funkcje zagnieżdżone

- Funkcje mogą się zagnieżdżać
- Funkcja zagnieżdżona ma dostęp do zmiennych funkcji zewnętrznej

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

10

15

15

15

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    add()
    return y
}
returnFifteen()
```

10

(2 times)

20

20

Funkcje są typem pierwszej klasy

- Funkcja może być traktowana jak inne typy
- Funkcje mogą przekazywać funkcję

```
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

```
8

(Function)

(Function)
8
```

- i być przekazywane jako argument

```
func hasAnyMatches(list: [Int], condition: Int -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, lessThanTen)
```

```
true

(3 times)

[20, 19, 7, 12]
true
```

Funkcje i domknięcia

- Funkcje są szczególnymi przypadkami domknięć (closure)
- Domknięcia to segmenty kodu, które mogą być wywoływane wielokrotnie
- Domknięcie jest tworzone pomiędzy nawiasami klamrowymi { }

```
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

```
[60, 57, 21, 36]

(4 times)
(4 times)
```

```
let mappedNumbers = numbers.map({ number in 3 * number })
println(mappedNumbers)
```

```
(5 times)
"[60, 57, 21, 36]"
```

wersja uproszczona

Obiekty i klasy

- Klasa jest deklarowana słowem kluczowym class
- Wszystko zdefiniowane wewnątrz klasy jest jej elementem składowym (pola, metody, własności)

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

```
{numberOfSides 0}
{numberOfSides 7}
"A shape with 7 sides."
```

Konstruktor klasy

- Metoda zaczynająca się od `init`, nie posiadająca typu
- Jeżeli obiekt jest kosztowny może posiadać destruktor `deinit`

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

Dziedziczenie

- Tylko pojedyncze
- Składnia jak w C++/C# - klasa nadrzędna po **:**

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() ->  Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

```
27.04

"A square with sides of length 5.2."

{{numberOfSides 4 name "my test square"}
27.04
"A square with sides of length 5.2."
```

Przeciążanie metod Własności

- Tylko ze słowem kluczowym override
- Własności mogą mieć getter i setter

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
        get {
            return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0
        }
    }

    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \
            (sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
println(triangle.perimeter)
triangle.perimeter = 9.9
println(triangle.sideLength)
```

klasa nadrzędna

nowa wartość

```
9.3

{{numberOfSides 3 name "a triangle"} side...

{{numberOfSides 3 name "a triangle"} side...
"9.3"
{{numberOfSides 3 name "a triangle"} side...
"3.3"
```

Własności

- Jeżeli własność nie wymaga obliczania, ale wymaga sprawdzenia przed lub po zmianie, można wykorzystać sekcje `willSet` i `didSet`

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength = newValue.sideLength
        }
    }
    var square: Square {
        willSet {
            triangle.sideLength = newValue.sideLength
        }
    }
    init(size: Double, name: String) {
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength: size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test
    shape")                                                    {{{...} sideLength 10.0} {...} sideLength 10....
println(triangleAndSquare.square.sideLength)                   "10.0"
println(triangleAndSquare.triangle.sideLength)                 "10.0"
triangleAndSquare.square = Square(sideLength: 50, name: "larger square")  {{{...} sideLength 50.0} {...} sideLength 50....
println(triangleAndSquare.triangle.sideLength)                 "50.0"
```

{{numberOfSides 3 name "another test sha...

Typy wyliczeniowe

- Deklarowane przez słowo kluczowe `enum`

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace"
        case .Jack:
            return "jack"
        case .Queen:
            return "queen"
        case .King:
            return "king"
        default:
            return String(self.rawValue)                    (2 times)
        }
    }
}
let ace = Rank.Ace                                          (Enum Value)
let aceRawValue = ace.rawValue                              1
```

typ bazowy

Typem bazowym może być `Int`, `String` lub typy zmiennoprzecinkowe

Typy wyliczeniowe

- Zmienną wyliczeniową można utworzyć z typu bazowego

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

"3"

- Typ wyliczeniowy może nie mieć typu bazowego

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
        case .Spades:
            return "spades"
        case .Hearts:
            return "hearts"
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()
```

"spades"

"hearts"

(Enum Value)
"hearts"

Typy wyliczeniowe

- składowe typu wyliczeniowego mogą mieć skojarzone ze sobą wartości
- mogą się one różnić w ramach jednego typu

```
enum ServerResponse {
    case Result(String, String)
    case Error(String)
}

let success = ServerResponse.Result("6:00 am", "8:09 pm")
let failure = ServerResponse.Error("Out of cheese.")

switch failure {
case let .Result(sunrise, sunset):
    let serverResponse = "Sunrise is at \(sunrise) and sunset is at \
        (sunset)."
case let .Error(error):
    let serverResponse = "Failure...  \(error)"
}
```

(Enum Value)
(Enum Value)

"Failure... Out of cheese."

Struktury

- Są podobne do klas
- Przekazywane zawsze przez wartość (klasy przez referencję)

- Słowo kluczowe `struct`

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \
            (suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

```
"The 3 of spades"

(Enum Value), (Enum Value)
"The 3 of spades"
```

Protokoły

- Odpowiednik interfejsów
- Słowo kluczowe `protocol`

metoda będzie zmieniać obiekt

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

- Mogą być adaptowane (implementowane) przez klasy struktury i wyliczenia

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

```
{simpleDescription "A very simple class." a...
{simpleDescription "A very simple class.  N...
"A very simple class.  Now 100% adjusted."

{simpleDescription "A simple structure"}
{simpleDescription "A simple structure (adj...
"A simple structure (adjusted)"
```

Protokoły

- Protokoły mogą być używane jak każdy inny typ

```
let protocolValue: ExampleProtocol = a
println(protocolValue.simpleDescription)
println(protocolValue.anotherProperty)  // Uncomment to see the error
```

{simpleDescription "A very simple class. N...
"A very simple class. Now 100% adjusted."

Rozszerzenia

- Dodawanie nowej funkcjonalności do istniejących klas
- Słowo kluczowe `extension`

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
println(7.simpleDescription)
```

"The number 7"

"The number 7"

Typy generyczne

- Podobnie jak w innych językach deklarowane za pomocą < >

- Generyczne mogą być funkcje, metody, klasy, typy wyliczeniowe i struktury

```
func repeat<Item>(item: Item, times: Int) -> [Item] {
    var result = [Item]()
    for i in 0..<times {
        result.append(item)
    }
    return result
}
repeat("knock", 4)
```

```
0 elements
(4 times)
["knock", "knock", "knock", "knock"]
["knock", "knock", "knock", "knock"]
```

```
// Reimplement the Swift standard library's optional type
enum OptionalValue<T> {
    case None
    case Some(T)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger = .Some(100)
```

```
(Enum Value)
(Enum Value)
```

Typy generyczne

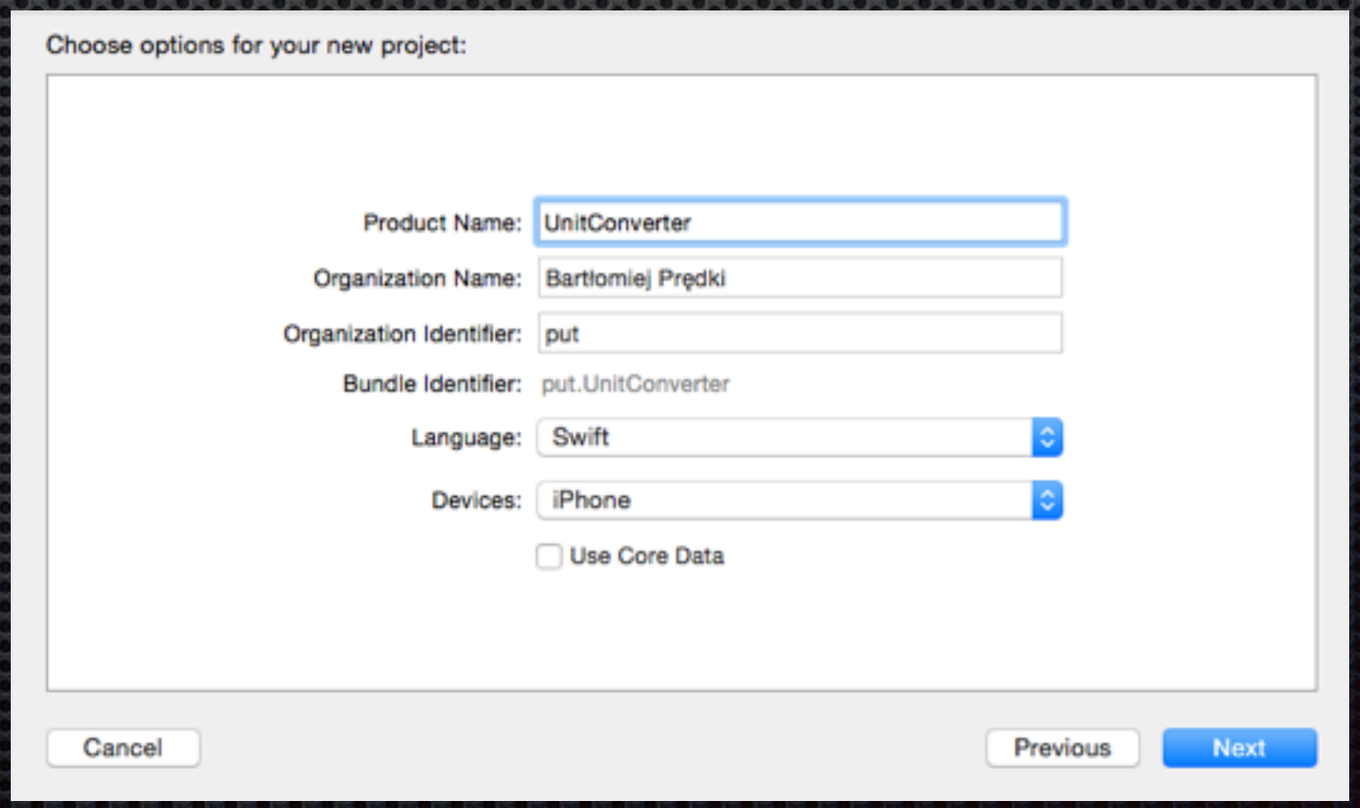
- Typ generyczny może stawiać wymagania, np. co do realizacji protokołów z użyciem słowa kluczowego `where`

```
func anyCommonElements <T, U where T: SequenceType, U: SequenceType, T.
    Generator.Element: Equatable, T.Generator.Element == U.Generator.
    Element> (lhs: T, rhs: U) -> Bool {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true                                          true
            }
        }
    }
    return false
}
anyCommonElements([1, 2, 3], [3])                                   true
```

Pierwsza aplikacja

Pierwsza aplikacja

- Prosty konwerter wartości ze stopni Celsjusza na Fahrenheita
- Szablon Single View Application



UnitConverter 〉 iPhone 6 | UnitConverter: **Ready** | Today at 12:35

UnitConverter
2 targets, iOS SDK 8.3
- UnitConverter
  - AppDelegate.swift
  - ViewController.swift
  - Main.storyboard
  - Images.xcassets
  - LaunchScreen.xib
  - Supporting Files
- UnitConverterTests
- Products

UnitConverter

UnitConverter | General | Capabilities | Info | Build Settings | Build Phases | Build Rules

Identity

Bundle Identifier: put.UnitConverter
Version: 1.0
Build: 1
Team: None

Deployment Info

Deployment Target: 8.3
Devices: iPhone
Main Interface: Main
Device Orientation: Portrait, Upside Down, Landscape Left, Landscape Right
Status Bar Style: Default
Hide status bar

App Icons and Launch Images

App Icons Source: AppIcon
Launch Images Source: Use Asset Catalog
Launch Screen File: LaunchScreen

Embedded Binaries

Add embedded binaries here

Linked Frameworks and Libraries

Name | Status

Add frameworks & libraries here

Identity and Type

Name: UnitConverter
Location: Absolute
UnitConverter.xcodeproj
Full Path: /Users/bpredki/Dysk Google/Projects/UnitConverter/UnitConverter.xcodeproj

Project Document

Project Format: Xcode 3.2-compatible
Organization: Bartłomiej Prędki
Class Prefix:

Text Settings

Indent Using: Spaces
Widths: 4 Tab, 4 Indent
Wrap lines

Source Control

Repository: UnitConverter
Type: Git
Current Branch: Branches Unavailable
Version: --
Status: No changes
Location: /Users/bpredki/Dysk Google/Projects/UnitConverter/UnitConverter.xcodeproj

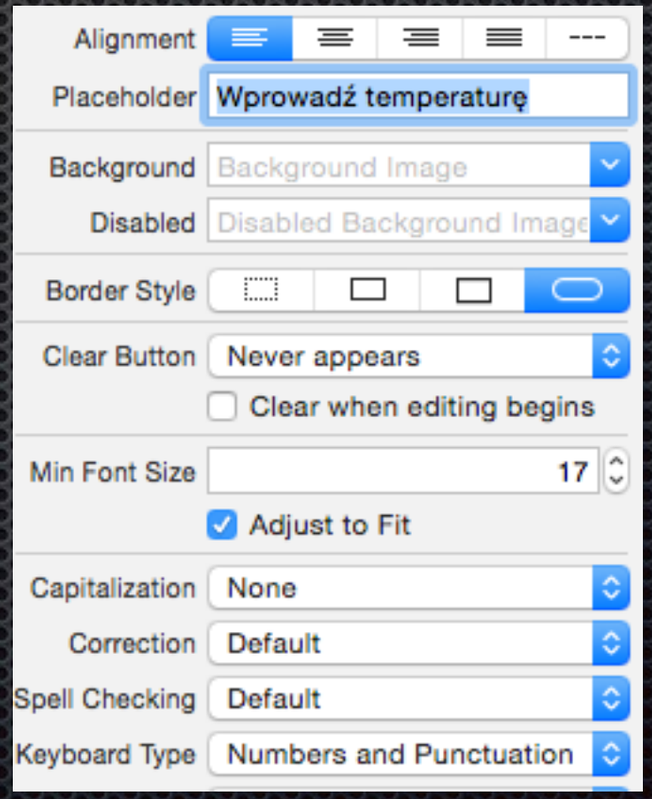
View Controller - A controller that supports the fundamental view-management model in iOS.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

Table View Controller - A controller that manages a table view.

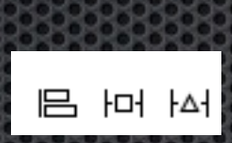
Konwerter

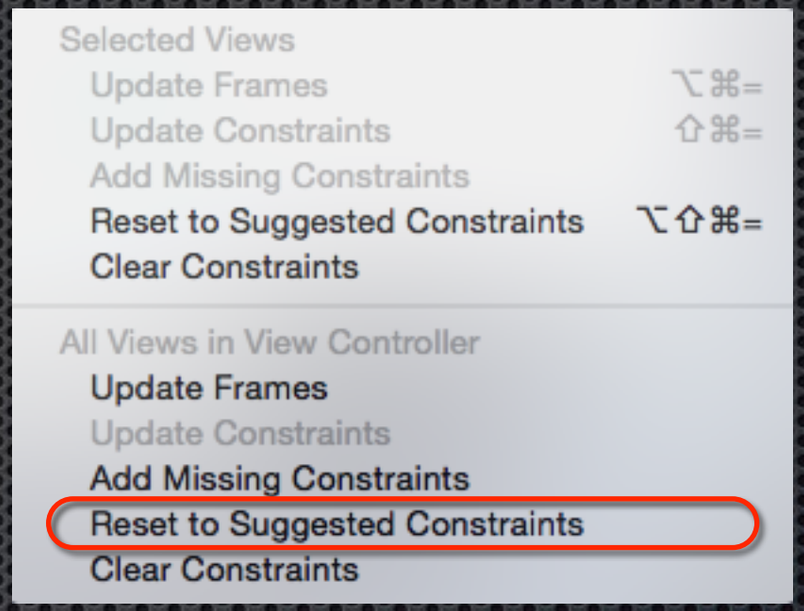
- Otwieramy scenorys
- Dodajemy pole tekstowe, guzik i etykietkę



Konwerter

- Ustawienia Auto Layout





Klasa AppDelegate

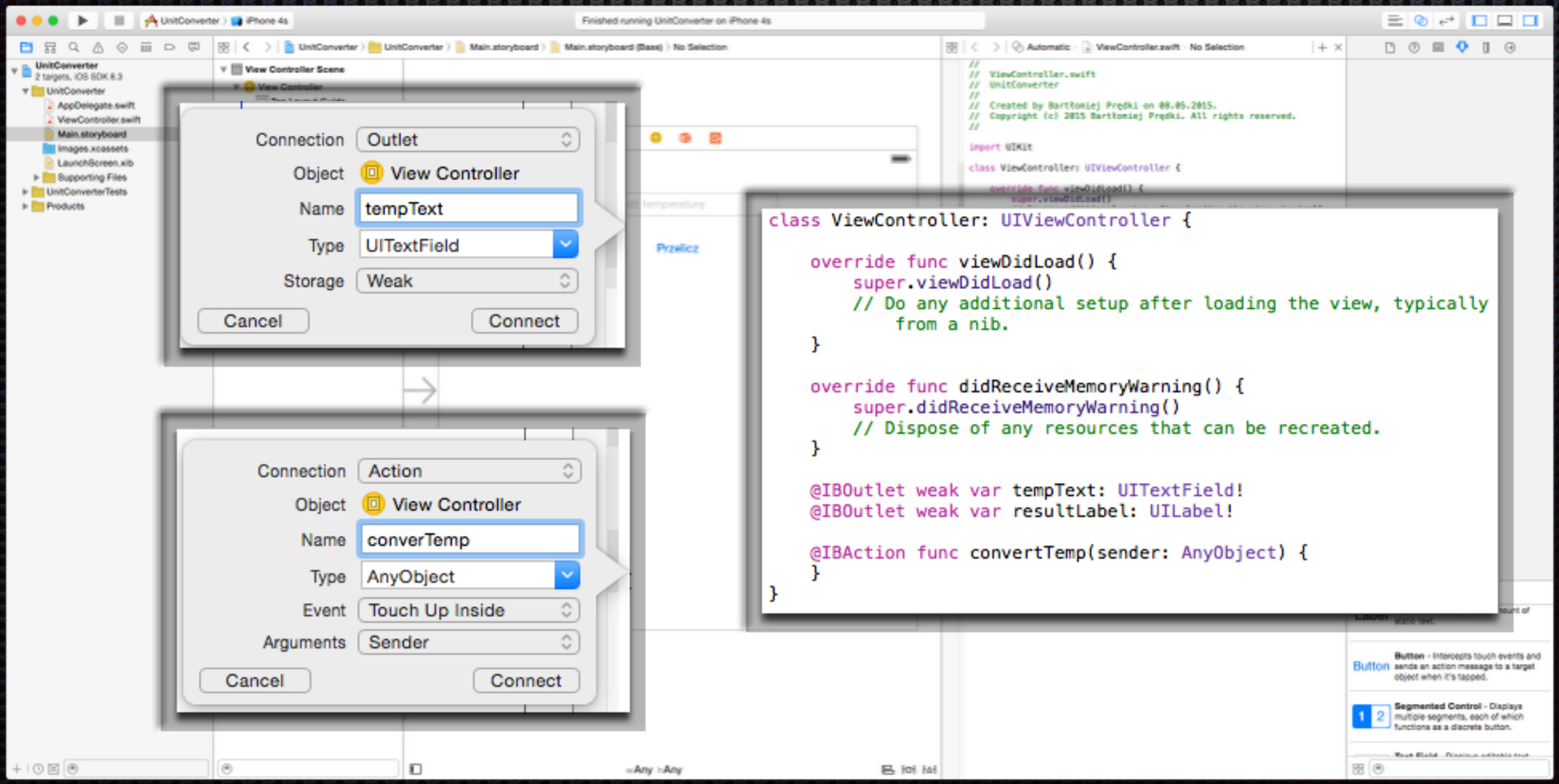
```swift
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    func applicationWillResignActive(application: UIApplication) {
        // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary
            interruptions (such as an incoming phone call or SMS message) or when the user quits the application and it begins the
            transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games should use this method
            to pause the game.
    }

    func applicationDidEnterBackground(application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate timers, and store enough application state information
            to restore your application to its current state in case it is terminated later.
        // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user
            quits.
    }

    func applicationWillEnterForeground(application: UIApplication) {
        // Called as part of the transition from the background to the inactive state; here you can undo many of the changes made on
            entering the background.
    }

    func applicationDidBecomeActive(application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously
            in the background, optionally refresh the user interface.
    }

    func applicationWillTerminate(application: UIApplication) {
        // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.
    }


}
```

Klasa ViewController

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }


}
```

Konwerter

- Tworzymy gniazdka



Konwerter

- Uzupełniamy metodę

```
@IBAction func convertTemp(sender: AnyObject) {
    let celsius = (tempText.text as NSString).doubleValue
    let fahrenheit = celsius * 1.8 + 32
    let resultText = "Fahrenheit \(fahrenheit)"
    resultLabel.text = resultText
}
```

- Dodajemy chowanie klawiatury

```
override func touchesBegan(touches: Set<NSObject>, withEvent event: UIEvent) {
    tempText.endEditing(true)
}
```

Konwerter

iOS Simulator - iPhone 4s - iPhone 4s / iOS 8.3 (12F69)
Carrier
1:04 PM
Game Center
Passbook
Extras
Apple Watch
UnitConverter
Safari